From the Theory to the Tools: Parallel Dynamic Programming

D. González, F. Almeida, J. Roda, C. Rodríguez

Centro Superior de Informática

Dpto. E.I.O. y Computación

Universidad de La Laguna, Tenerife, Spain

casiano@ull.es

Abstract. Dynamic Programming is an important paradigm that has been widely used to solve problems in various areas as control theory, operation research, biology and computer science. We generalize the Finite Automaton Formal Model for Dynamic Programming deriving pipeline parallel algorithms. The optimality of these algorithms is established for the new class of Non Decreasing Finite Automata. As an intermediate step for the construction of skeleton for the automatic parallelization of Dynamic Programming, we have developed a tool for the implementation of pipeline algorithms. The tool maps the processes in the pipeline in the target architecture following a mix of block and cyclic policies adapted to the grain of the machine. Based on the former tool, the automatic parallelization of Dynamic Programming is straightforward. The use of the model and its associated tools is illustrated with the Single Resource Allocation Problem. The performance and portability of these tools is compared with specific "hand made" code written by experienced programmers. The experimental results on distributed memory, and shared distributed memory architectures prove the scalability of the proposed paradigm and its associated tools.

Keywords Models, Tools, Dynamic Programming, Resource Allocation

1 Introduction

Dynamic programming (DP) is an important problem-solving technique that has been widely used in various fields such as control theory, operations research, biology and computer science. The Dynamic Programming technique is frecuently used when the solution to a problem can be viewed as the result of a sequence of decisions. Many of these problems need the whole enumeration of the decision sequences and then to pick the best. The time and space requirements may be prohibitive. Dynamic Programming often drastically reduces the amount of enumeration by avoiding to consider some decision sequences that cannot possibly be optimal. In Dynamic Programming an optimal sequence of decisions is arrived at by making explicit appeal of the principle of optimality.

Sequential computation for Dynamic Programming has been studied extensively. General discrete Dynamic Programming programs viewed as multistage finite automatons with a certain cost structure superimposed were presented in [12, 15]. Many authors describe parallel algorithms for specific Dynamic Programming problems. Parallel dynamic programming algorithms for 0/1, two-dimensional and integer knapsack problems have been proposed for different architectures in [1, 3, 4, 6, 16, 17, 20, 23, 25].

Specific recurrence relations such as shortest paths, optimal binary search tree, optimal chained matrix multiplication parenthesizations, etc. have been approached for systolic arrays [18, 22], general shared-memory multiprocessors [8] and for PRAM models [9, 10]. All these contributions were proposed for specific problems and specific architectures. The only proposal for a general parallel DP approach was due to Wah et al. in 1985 [26]. They suggest parallelizations in terms of a classification of the dynamic programming problems according to the form of the functional equation. The general idea of their work is that solving a dynamic programming problem can be reduced to finding the shortest paths in the associated graph. This problem can be solved by the iterative multiplication of matrices. From a practical point of view these matrix algorithms present, for the majority of cases, an unnecessary and prohibitive complexity: $O(n^3 logn)$, where n is the number of vertex in the graph.

Following Ibaraki's [12] discrete Dynamic Programming approach, we extend the sequential model for DP to a parallel model. We propose a general parallel dynamic programming algorithm for pipeline and ring networks for multistage automatons. The parallel algorithm presented constitutes a paradigm to solve DP problems and provides the conceptual framework for the development of skeleton tools. Based in this paradigm, we present a skeletton oriented tool for the implementation of Multistage Dynamic Programming algorithms. The level of this skeleton tool is even higher than the provided by high level languages as High Performance Fortran or Compositional C++, since they not only simplifie the writing of parallel programs but also the development of the sequential case. The tool presented derives a parallel DP program from the user specification of a sequential code. The tool manages the optimal mapping of the stages of the associated DP automata in the actual processors, the generation of the corresponding messages and the handle of the involved data structures.

To study the performance of our skeleton, experiments were carried out in three different architectures: a distributed memory computer, the IBM-SP2; a distributed-shared memory computer, the Origin 2000; and a distributed memory with shared address space computer, the Cray T3E. The SP2 from IBM has 44 processors, 12 GB main memory, 431 GB disk storage and 18.55 Gflop/s theoretical peak. The Origin 2000 from Silicon Graphics is a 64 R10000 processors (196 MHz) machine with 8 GB main memory, 288 GB disk and 25.08 Gflop/s theoretical peak. The Cray T3E has 32 DEC 211164 processors. Each processor has 128 Mb of memory and reaches 600 Mflops.

Section 2 introduces a formal approach for Dynamic programming based in the automata theory. The model is illustrated with the Single Resource Allocation Problem. A general sequential algorithm is derived from the model. Section 3 presents the class of Non Decreasing Finite Automata, for which an optimal pipeline algorithm is delivered. Tools for automatic parallelization of DP will be described in section 4. The efficiency of these tools is compared with specific "hand made" code written by experienced programmers. The experimental results for the Single Resource Allocation Problem prove the scalability of the proposed paradigm and its associated tools in these three significant parallel architectures. The performance measurements obtained by the tool show no essential loss of efficiency, when compared with a message passing implementation in MPI or PVM.

2 The Dynamic Programming Technique

Dynamic Programming is an important technique for solving combinatorial optimization problems. It is based in the principle of optimality introduced by R. Bellman [5]. Our goal is to solve the optimization problem:

$$\begin{aligned} &minimize\ (maximize)\ f(x) \\ &subject\ x \in S \end{aligned} \tag{1}$$

in which the set of constraints S can be modeled as a language, that is, $S \subseteq \Sigma^*$. The set Σ is a finite set called the set of decisions. The elements of Σ^* which are elements of the Kleene closure of Σ, are referred or called policies. A policy $x \in \Sigma^*$ is said to be feasible if $x \in S$.

The Dynamic Programming technique works by associating to the optimization problem an automaton with costs [12]:

Definition: An automaton with costs is a triple $\prod = (M,\ \mu,\ \xi_0)$, where:

1) M is a finite automaton $M = (Q,\ \Sigma,\ q_0,\ \lambda,\ Q_F)$ [11] in which:

Q denotes the set of states,

Σ is the set of decisions,

$q_0 \in Q$ is the initial state,

$\lambda : Q \times \Sigma \to Q$, is the transition function and

$Q_F \subset Q$ is the set of final states.

The transition function (λ) can be extended to the set of policies; $\lambda : Q \times \Sigma^* \rightarrow Q$ by defining $\lambda(q, \epsilon) = q$ and $\lambda(q, xa) = \lambda(\lambda(q, x), a)$, where $q \in Q$, $x \in \Sigma^*$, and $a \in \Sigma$. The notation $\lambda(x) \cong \lambda(q_0, x)$ for all $x \in \Sigma^*$ will be used. There is no risk of confusion between the two uses of λ, since the number and the type of the arguments allow the reader to differentiate them. The same consideration holds for the following notation: $\lambda(q) = \{q' \in Q \textit{ such that exists } a \in \Sigma : \ \lambda(q, a) = q'\}$.

2) $\mu : \Re \times Q \times \Sigma \rightarrow \Re$ is the cost function, where $\Re$ denotes the set of real numbers. If decision a $\in \Sigma$ is applied to state $q \in Q$, and the accumulated cost is ξ, the automaton changes into the new state $\lambda(q, a)$ with new cost given by $\mu(\xi,\ q,\ a)$. The cost function μ can be extended to $\Re\ \times Q\ \times \Sigma^* \rightarrow \Re$ by defining $\mu(\xi, q, xa) = \mu(\mu(\xi, q, x),\ \lambda(q, x), a)$ for $\xi \in \Re$, $q \in Q$, $x \in \Sigma^*$, $a \in \Sigma$ and $\mu(\xi, q, \epsilon) = \xi$.

3) $\xi_0 \in \Re$ is the setup cost of the initial state q_0.

We will use notation $\mu(x)$ to denote the cost of policy x starting in q_0, i.e.: $\mu(x) \cong \mu(\xi_0, q_0, x)$ and $\rho(q)$ denotes the cost of the best policy starting at q, $\rho(q) = \min\{\mu(G(q), q, x)\ /\ \lambda(q, x) \in\ Q_F,\ x \in \Sigma^*\}$. We will also define $d^-(q)$ as the number of states transiting to q, that is $d^-(q) = |\lambda^{-1}(q)|$ and $d^+(q) = |\lambda(q)|$.

Definition: A finite automaton with costs $\prod = (M,\ \mu,\ \xi_0)$ represents the optimization problem (1) if and only if it satisfies:

1) The language accepted by the automaton M is the set S of feasible solutions, i.e.: $L(M) = \{x \epsilon \Sigma^* \ / \lambda(x) \epsilon Q_F\} = S$.

2) For all feasible policy $x \epsilon S$, $\mu(x) = f(x)$.

Definition: A feasible policy $x \epsilon S$ is said to be optimal if and only if $\mu(x) \leq \mu(y)$ for any other feasible policy $y \epsilon S$.

Along this work we will assume that function μ is monotone in ξ, that is, $\mu(\xi, q, a) \geq \mu(\xi', q, a)$ if $\xi \geq \xi'$. Furthermore we will also assume that it takes positive values $\mu(\xi, q, x) \geq 0$.

Definition: For any state $q \epsilon Q$ we define $G(q)$ as the optimal value of any policy leading from the start state q_0 to state q, i.e., $G(q) \cong \min\{\mu(x) \ / \ x \epsilon \Sigma^*, \lambda(x) = q\}$ when exist an $x \epsilon \Sigma^*$ such that $\lambda(x) = q$, and $G(q) = +\infty$ otherwise.

Let be G^* the optimum of the values for the feasible policies, that is $G^* = \min\{G(q) / \ q \epsilon Q_F\}$. Obviously, to solve the optimization problem (1) is equivalent to finding the optimal value G^*. It can be proved [12] that if $\prod = (M, \mu, \xi_0)$ represents problem (1) then, $G^* = \min\{f(x) \ / \ x \epsilon S\}$.

It can also be proved [12] that, the optimal value $G(q)$ can be obtained from the optimal values of the states from wich q is reacheable through some decision a, i.e., for all $q \neq q_0$, $G(q) = \min\{\mu(G(q'), q', a) \ / \ q' \epsilon Q, a \epsilon \Sigma, \lambda(q', a) = q\}$ and $G(q_0) = \xi_0$.

The former recursive formula gives the base for the Dynamic Programming algorithm.

Definition: Let be M a finite automaton, M is a multistage automaton if, and only if, there is a partition of the set of states such that the dependences occur only among states belonging to consecutive sets of the partition, i.e., there is a partition of the set Q, where $Q = \{q_0\} \cup Q_1 \cup ... \cup Q_n \cup Q_F \cup \{q_d\}$ such that $Q_i \cap Q_j = \emptyset$ when $i \neq j$, and:

$\lambda(q_0, a) \;\epsilon\; Q_1 \cup \{q_d\}$, for all $a \;\epsilon \Sigma$.

$\lambda(q, a) \;\epsilon\; Q_{i+1} \cup \{q_d\}$, for all $q \epsilon Q_i$, a $\epsilon \Sigma$, $i = 1, 2, .., n-1$.

$\lambda(q_d, a) = q_d$, for all $a \;\epsilon\; \Sigma$.

$\lambda(q, a) \;\epsilon\; Q_F \cup \{q_d\}$, for all $q \;\epsilon\; Q_n$ and for all $a \;\epsilon\; \Sigma$, where $q_d \;\epsilon\; Q - Q_F$ is called the dead state.

We will denote by $\Sigma(q)$ the set $\{b \;\epsilon\; \Sigma \;/\; \lambda(q, b) \neq q_d\}$.

The sequential dynamic programming algorithm in figure 1 straightforwardly solves the optimization problem (1) for multistage automatons with costs. Note that to obtain $G(q)$ we are assuming that it is necessary to compute in run time the states q' on which it depends.

Theorem 2.1: The complexity of algorithm in figure 1 is $O(\mid Q \mid \cdot \mid \sum \mid \cdot O(\mu))$.

proof: It is a trivial consequence of the fact that every state of the automaton must be traversed and, for each state, the cost function must be evaluated on each feasible transition.

Through this work we will assume a constant time for the computation of function μ. For those cases not satisfying this assumption the complexities can be obtained by simply multiplying by the factor $O(\mu)$ the complexities presented along the paper.

```
Begin
  G(q_0) = ξ_0;
  for k = 1 to n do
  begin
    for q ϵ Q_k do
       G(q) = ∞;
    for a ϵ ∑ do
       for q' ϵ Q_{k-1}
       begin
         q = λ (q', a);
         G(q) = min{G(q), μ(G(q'), q', a};
       end;
  end;
End.
```

Fig. 1. Sequential Dynamic Programming Algorithm.

2.1 An example: the Single Resource Allocation Problem (RAP)

The Single Resource Allocation Problem can be stated as follows [2, 13]:

$$
\begin{aligned}
maximize\ z &= \textstyle\sum_{j=1}^{N} f_j(x_j) \\
subject\ to \textstyle\sum_{j=1}^{N} x_j &= M
\end{aligned}
\tag{2}
$$

Namely, it is required to allocate M units of an indivisible resource to N activites so that the sum of the effectiveness measured by $f_j(x_j)$ is maximized.

In order to solve this problem by dynamic programming, we introduce the following states $q_{k,m}$ and their values $G(.)$.

$q_{0,0}$: The initial state representing the null allocation.

$q_{k,m}$: The state representing k-dimensional allocation vectors $(x_1, x_2, ..., x_k)$ of nonnegative integers such that $\sum_{j=1}^{k} x_j = m$, for $k = 1, 2, ..., N$, $m = 0, 1, ..., M$.

$q_{N,M}$: The final state representing the allocation vectors satisfying (2).

$G(q)$: The maximum of $\sum_{j=1}^{k} f_j(x_j)$ for the allocation vectors represented by state q.

The transition function can be defined as: $\lambda(q_{k,m}, x) = q_{(k+1),(m+x)}$ $x = 0, ..., M - m$

By definition, $G(q_{N,M})$ is the optimum objective value we want to compute. It is equal to the length of a longest path from the source $q_{0,0}$ to the sink $q_{N,M}$ where each edge $(q_{(k-1),m}, q_{k,(m+x_k)})$ has length $f_k(x_k)$. The value $G(q_{k,m})$ can be recursively computed by

$G(q_{0,0}) = 0$

$G(q_{1,m}) = f_1(m), m = 0, ..., M$

$G(q_{k,m}) = \max_{0 \leq x_k \leq m}\{G(q_{(k-1),(m-x_k)})+f_k(x_k)\}, k = 2, 3, ..., N, m = 0, 1, ..., M.$

3 Parallelization of the Dynamic Programming Technique

Definition: A multistage automaton M is called a non decreasing automaton if and only if there is a total order for each stage, $Q_k = \{q_{k,1}, ..., q_{k,m}\}$, such that for every $q_{k,i}$ on stage Q_k, the three following conditions are satisfied:

1) $\lambda(q_{k,i}) \subset [q_{(k+1),i},\ q_{(k+1),r}]$, where $r = |Q_{k+1}|$

2) $|\lambda(q_{k,i})| \leq |\lambda(q_{(k+1),i})|$

3) There is a constant C such that $|\{a \ \epsilon \ \Sigma \ \lambda(q_{(k-1),i},\ a) = q_{k,i}\}| \leq C$.

Note that the condition $|\lambda(q_{k,i})| \leq |\lambda(q_{(k+1),i})|$ is equivalent to $d^+(q_{k,i}) \leq d^+(q_{(k+1),i})$.

Among the examples of non decreasing automatons we can mention the DP formulations of the 0-1, integer and non linear knapsack problems [20, 21], resource allocation problems [13] and some scheduling problems [24].

Along the rest of this paragraph we will consider that the automaton M in $\prod = (M, \mu, \xi_0)$ is a non decreasing automaton.

Lemma 3.1: $|Q_k| \leq |Q_{k+1}|$ for any nondecreasing automaton M.

proof: Lets assume that exist k such that $|Q_k| > |Q_{k+1}|$, let r' and r denote $|Q_k|$ and $|Q_{k+1}|$ respectively. By property 1) of a non decreasing automaton, $\lambda(q_{k,r'}) \subset [q_{(k+1),r'},\ q_{(k+1),r}]$ which is not possible since $r' > r$. Therefore $|Q_k| \leq |Q_{k+1}|$ for any nondecreasing automaton.

Since we are dealing with a multistage automaton we can exploit the property that, for all $q \in Q_k$ the set $\{q' /\ \lambda(q',\, a) = q\}$ is included in Q_{k-1}. This fact allows to solve the problem using a pipeline with as many processors as stages by allocating the computation of optimal values in stage k, $G(q)$, where $q \in Q_k$, to processor k (Algorithm PAPS of figure 2). In many cases, processor k has enough information about the stage $k-1$ to build the states of Q_{k-1} in constant time. That is, it knows the number of states in the previous stage $|Q_{k-1}|$ and the order in which they will be computed and received. This order has to be the natural order given in Q_{k-1} by the non decreasing automaton. During the *ith* iteration processor k receives from processor $k-1$ the optimal value $G(q_{k-1})$. As soon as $G(q_{k-1})$ is received, the values $G(q)$ for those q such that exist $a \in \Sigma(q_{k-1})$ with $\lambda(q_{k-1},\, a) = q$ are updated. Property 1 of a non decreasing automaton grants that all the optimal values from which state $q_{k,i}$ depends on have been received and so the optimal value $G(q_{k,i})$ is completely computed and can be sent to processor $k+1$. The algorithm stops when processor n computes the optimal values for the final states $q_f \in Q_F$. Processor n holds the optimal solution G^*.

---

begin
  for $q_{(k-1),i} \in Q_{k-1}$ do
  begin
    ReceiveFromLeft $(G(q_{(k-1),i}))$;
    for $a \in \sum$ such that $\lambda(q_{(k-1),i},\ a) \ = \ q_{k,i}$ do
        $G(q_{k,i}) = \min\{G(q_{k,i}),\ \mu(G(q_{(k-1),i}),\ q_{(k-1),i},\ a)\}$;
    SendToRight $(G(q_{k,i}))$;
    for $q_{k,j} = \lambda(q_{(k-1),i},\ a)$ with $a \in \sum(q_{(k-1),i})$ do
        $G(q_{k,j}) = \min\{G(q_{k,j}),\ \mu(G(q_{(k-1),i}),\ q_{(k-1),i},\ a)\}$;
  end;
end.

Fig. 2. A Pipeline Algorithm Parallelizing on the Stages for non decreasing automatons (PAPS). Code for processor k.

Next, we will establish the conditions to guarantee an optimal speedup for the proposed algorithm under the former assumption.

Lemma 3.2: In the algorithm PAPS, processor n, in charge of the last stage, starts to work at iteration $O(n)$.

proof: Obviously, the first processor sends the first optimal value in time $O(d^-(q_{1,1})) = O(1)$, which is the time needed to compute $G(q_{1,1})$. Then, the value $G(q_{1,1})$ is received by processor 2 at the next iteration, i.e., at iteration two. Processor 2 computes $G(q_{2,1})$ and sends it to processor 3, which will receive it at the third iteration. By induction, processor k receives at iteration k the first optimal value $G(q_{(k-1),1})$. State $q_{k,1}$ is completely updated after the completation of the $O(d^-(q_{k,1}))$ iterations of the loop in lines 5-6. Condition 3) of the non decreasing automaton definition implies $O(d^-(q_{k,1})) = O(1)$. Thus, processor $k+1$ receive the first value $G(q_{k,1})$ in time $O(k)$.

In many cases the special characteristics of the considered problem allow a clever strategy leading to an implementation of the loop 8-9 with complexity $O(|\lambda(q_{(k-1),i})|)$. To accomplish it the programmer should be able to obtain each decision from the set $\sum(q_{(k-1),i})$ and each state depending on it in constant time. As an example see line 8 in figure 4.

Lemma 3.3: Processor k of the pipeline spends just a constant time at line 4 of the algorithm of figure 2 waiting for values of the left neighbor $k-1$.

proof: The lemma is trivial for the first processor. Let us assume that processor $k-1$ waits a constant time for the values of its left neighbor and let us see that processor k waits a constant time. After sending the value $G(q_{(k-1),i})$ (to avoid delays in the output of line 7 we assume, if necessary, the presence of buffers) processor $k-1$ goes into the loop of line 8-9. This loop has a complexity less than $O(|\lambda(q_{(k-2),i})|)$. Meanwhile, processor k is traversing the two loops of lines 5-6 and 8-9 plus the sending of line 7. This leads to a complexity $O(|\lambda(q_{(k-1),i})|)$. Since M is a non decreasing automaton $|\lambda(q_{(k-2),i})| \leq |\lambda(q_{(k-1),i})|$, and so processor $k-1$ sends the optimal value $G(q_{(k-1),(i+1)})$ to processor k after completing the loop of lines 5-6 in $O(1)$. Thus the lemma holds.

Theorem 3.1: Let $\prod = (M, \mu, \xi_0)$ be a non decreasing automaton with costs and let be $m = |Q_n|$, then the algorithm PAPS has complexity: $O(n+\sum_{i=1}^{m} d^-(q_{n,i}))$.

proof: As a consequence of the lemma 3.2, processor n starts to work on time $O(n)$ and since according to lemma 3.3 no delay due to communication occurs, then it spends $O(\sum_{i=1}^{m} d^-(q_{n,i}))$ with the computation.

The following theorem states that if time needed to compute the last stage of the automaton has the same order than the time invested by the sequential algorithm divided by the number of processors on the pipeline, then the non decreasing automaton can be parallelized in optimal time.

Theorem 3.2: Let $\prod = (M, \mu, \xi_0)$ be a non decreasing automaton with costs such that exists a constant D holding the condition:

$$D(\sum_{k=1}^{n} \sum_{q_{ki} \epsilon Q_k} d^-(q_{k,i}))/n = \sum_{i=1}^{m} d^-(q_{n,i})$$

under this assumption the algorithm PAPS is optimal.

proof: It is a direct consequence of theorem 3.1, theorem 2.1 and the fact that the hypothesis of the theorem guarantees an optimal load balancing.

4 Tools for Parallel Dynamic Programming

Most of the efficient parallel techniques for solving DP [1, 2, 3, 4, 6, 20] exploit pipeline parallelism. These kind of algorithms, as in figure 2, require a variable number of processors, usually proportional to the number of stages or the number of states per stage. They do not consider the fact that for most cases the number of processors required exceeds the number of available processors. Therefore, it is necessary to assign the processes among the available processors.

This is the problem of finding an efficient mapping of the virtual pipeline on the actual parallel machine. Fortes and Moldovan propose in [22] to partition the set of processes in sets (or bands) $B(i)$ of the same cardinality as the number p of processors, in such a way that process $q \in B(i)$ if and only if $q/p = i$. Process q is processed by processor $q \ mod \ p$. All the processes in the same set $B(i)$ are processed in parallel. Implementation can be easily achieved on a one way ring topology where the first and last processors are connected through a buffering process. This approach leads to a pure cyclic mapping. Better results depending on the grain g of the underlying platform may be achieved if this cyclic mapping is combined with block mapping. On a block-cyclic mapping, process q is assigned to processor $(q/g) \ mod \ p$. We aim to build a skeleton for parallel dynamic programming based in the methodology exposed in the former paragraph. Our main objectives are: Easiness of use, Portability and Efficiency. To achieve these goals the project is divided in two stages: the first one is the construction of a tool (called *La Laguna Pipeline*, *llp*) providing the programmer with a virtually infinite pipeline. The programmer just has to take control of the behaviour of a general cell of the pipe. The *La Laguna Pipeline* simulates the total amount of processors and maps them onto the physical ones using a mix of block and cyclic policies adapted to the grain g of the target architecture. The second one is properly the construction of the parallel dynamic programming skeleton or frame (called *La Laguna Parallel DP* or *llpdp*), built in top of the *llp*.

4.1 The basic tool: Giving Support for Pipeline Paralellism

La Laguna Pipeline is a general purpose tool for the pipeline programming paradigm, and it is not specifically conceived for Dynamic Programming. The portability of the tool is guaranteed by the wide number of message passing libraries supported (MPI, PVM, InmosC). The intensive use of macros instead of subroutines contributes to diminish the overhead introduced by the tool and enlarges its efficiency, as will be proved in the computational results in this section. Figures 3, 4 and 5 illustrates the easiness of use applying the general pipeline scheme presented in figure 2 to the Single Resource Allocation Problem (RAP) described in section 2.1.

```
void solve_RAP (void) {
  int  result;
  if (NAME == 0)
    ini_stage();
  else if (NAME == (N - 1)) {
    result = last_stage();
    GPRINTF("\ n % d:Result = % d",NAME,result);
  }
  else
    transition();
}
```

Fig. 3. First and Last Processors run different code.

The call to the macro $PIPE$ in figure 5 has as arguments the generic routine, $solve\_RAP()$ and the size of the pipeline N. This macro introduces the $solve\_RAP()$ routine in a loop simulating the virtual processors assigned to the actual processor according to the mapping policy. The variable $NAME$ identifies

```
void  transition ()
{
int m, j, x;
for( m = 0; m ¡= M; m++)
    G[m] = 0;
for (m = 0; m ¡= M; m++) {
    IN(&x);
    G[m] = max(G[m], x + f(NAME-1, 0));
    OUT(&G[m], 1, sizeof(int));
    for (j = m + 1; j ¡= M; j++)
      G[j] = max(G[j], x + f(NAME - 1, j - m));
} /* for m ... */
} /* transition */
```

Fig. 4. Simple Pipeline Algorithm for the RAP. Code for Processor NAME.

```
int  main (PARAMETERS)
{ INITPIPE;
  if (ARGC != 3) {
     abort();
  }
  N = atoi(ARGV(1)); /* Number of activities */
  M = atoi(ARGV(2)); /* Units of resource */
  PIPE(solve_RAP(), N);
  EXITPIPE;
}
```

Fig. 5. Function main(): expanding a PIPE of N virtual processors.

the virtual processor in the pipeline. Macros IN and OUT establish the corresponding communications with the left and right neighbors. *La Laguna Pipeline* provides parallel Input/Output routines as the $GPRINTF$ call in figure 3, allowing processors to read and write from shared input and output streams.

```
1: #define PIPE(f, n) {
      /* Calculate the number of STAGES to compute */
2:    int bands = numbands(n, g);
3:    for (i = 0; i < bands; i++) {
        /* Physical Processor pname calculates the NAME of the first process to simulate */
4:      NAME = g * (pname + i * numproc);
5:      if (NAME ≤ (n - 1))
6:        f;
7:    }
8: }
```

Fig. 6. The Macro PIPE.

4.2 Implementing Cyclic and Blocking-Cyclic Mapping Policies

The implementation of a pure cyclic mapping policy for a pipeline can be easily obtained over a ring as a particular case of the macro PIPE described in figure 6 taking $g = 1$. Every processor calculates the number of STAGES to compute, (n/p) (line 2), and then goes into a loop running the function f for the appropiated first virtual processor NAME (line 4).

The classical technique to obtain block-cyclic mapping consisting of g sequential executions of the f function is not a good approximation. The delay introduced by each processor produces a parallel algorithm as slow as the sequential one. To obtain an efficient implementation, processors must start to work as soon as possible and they must be feeded with data when needed. To fulfil this conditions we implement the block-cyclic mapping using the Unix standard libray "*setjmp.h*". This libray allows unconditional jumps to variable labels. To execute function f in pipeline with grain g, automatically g copies of the local data of function f are expanded, one for each one of the processes to be executed in one iteration of the loop. The code to be executed is the same for each process: the function f. Therefore, only the Program Counter of every process need to be stored. Macros IN and OUT differentiate between internal and external communications. External comunications use the communication primitives provided by PVM/MPI. Internal comunications produce the context switch between processes using the functions *setjmp*() and *longjmp*() of the libray "*setjmp.h*". The context switch just consist of updating the NAME of the process, and to get the Program Counter and the address of the local variables for this process. Since a context switch occurs only when needed and it is implemented very efficiently, the overhead introduced by the tool in a block-cyclic mapping is minimum.

4.3 Performance of llp

To measure the performance of the llp tool, we use the Dynamic Programming algorithm for the Resource Allocation Problem described in previous sections. The test problem considers N = 400 tasks and M = 1000 units of resource. Table 1 shows the times of the sequential algorithm. In all tables, columns are labeled with the three considered architectures and the times are expresed in seconds. The computational experience compares the results obtained with *llp* against the same algorithm written by an experienced programmer using the message passing library PVM/MPI (subcolumn labeled (ad-hoc). Rows are labeled with the number of processors. Each entry shows the time and the speedup obtained. Table 2 shows the comparison *llp* versus ad-hoc using cyclic partitioning and tables 3 and 4 give the results for *llp* and ad-hoc code using block-cyclic mapping respectivelly using different different values for the grain g (column labeled

Grain).

For the cyclic case (Table 2) the ad-hoc column presents better times than the corresponding *llp* times, as can be expected. The performance measurements obtained by the tool show no essential loss of efficiency, when compared with a message passing implementation in MPI or PVM. The loss of efficiency LE = (ad-hoc $-$ llp)/ad-hoc is 20% in the IBM-SP2, 15% in the Cray T3E and only 3% in the Origin, when the worst case is considered. The difference on the performances are explained by the differences in the architectures. As the theoretical results predicted, the algorithm presents a good scalabitity with increasing speedups for all the architectures.

For the block-cyclic case (Tables 3 and 4) it must be said that, to obtain fare comparisons, the optimization level of the compiler should be considered. The typical ad-hoc implementations of block-cyclic mapping are achieved through a loop that traverses the first state of every stage asigned to the processor, then traverses the second state of every stage and so on. The states are usually stored in a matrix. The *llp* tool does not implement a loop directly but uses the "*setjmp.h*" library. The states are stored in vectors, one vector per stage as a consequence of the automatic replication of the stages. When the optimization level used at compilation is the minimum, the loop of the ad-hoc algorithm is not optimized and runs slower than the *llp* tool due to the effects produced by a non efficient use of the caches. That is what happens with the IBM-SP2 and with the Origin where the minimum level of optimization was used. In both architectures, anomalous super-efficiency appears as a consequence of this phenomenon. In the Cray T3E the maximum level of optimization was used in the compiler so, the loop of the ad-hoc algorithm is optimized and runs faster. In any case, the behaviour of *llp* is comparable to the behaviour of the ad-hoc code.

An analysis of the grain allows to say that when the number of processors is increased the grain must be reduced to decrease the contention of initialization on each band $B(i)$. In the CRAY T3E, this effect is less visible due to the more optimized code used.

The only overhead introduced by the tool is the context switch through calls to jumps functions that is quite low and the overhead introduced in macros IN and OUT, that are designed to send and receive data of any type. This macros introduce additional information in the packets of data to be sent by the user.

Table 1. Sequential time of DP algorithm

IBM SP2	Origin 2000	Cray T3E
88.29	28.47	19.60

4.4 The Dynamic Programming Skeletton

Sequential dynamic programming algorithms (figure 1) follow a bread-first-search in which first stages of the automaton are visited in first place. It is well known that this kind of traversing implies the use of a FIFO queue. The *llpdp* dynamic programming skeleton takes advantage of the fact that a user can not distinguish the behaviour of the former pipeline from a FIFO queue. This way, the *llpdp* user

Table 2. Times and Speedup for llp versus ad-hoc code.

	IBM-SP2				O2000				CRAY T3E			
Proc.	ad-hoc		llp		ad-hoc		llp		ad-hoc		llp	
	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
2	61.47	1.44	71.86	1.23	40.90	0.70	40.54	0.70	22.07	0.89	24.18	0.81
4	30.97	2.85	36.15	2.44	20.52	1.39	20.88	1.36	11.35	1.73	12.68	1.55
8	15.85	5.57	18.41	4.79	10.30	2.76	10.55	2.70	6.12	3.20	6.83	2.87
16	7.98	11.06	9.27	9.52	5.29	5.37	5.44	5.23	3.15	6.23	3.60	5.44

is asked to write the sequential code for a generic automaton stage using a FIFO queue provided by the *llpdp* system. Instead of speaking of processors NAMEs the *llpdp* uses STAGES and INs and OUTs are substituted by INSERTs and REMOVEs. This simply renaming and semantic change means that the code generated is exactly the same that the generated by the *llp* tool for the code in figure 4. The parallel code generated is optimal when the automaton is Non Decreasing and the user inserts the states following the natural order of the automaton. Obviously, the efficiency achieved by the tool is equal to the obtained by the *llp* programming system.

5 Conclusions

We give a general parallel dynamic programming procedure that can be applied to a wide range of problems. The proposed scheme admit an easy and efficient implementation for the class of multistage automaton described as confirm the computational experiments in [2, 3, 20]. These computational experiments agree with the predicted theoretical results. The proposed algorithms constitute a

Table 3. Times and Speedup for adhoc code using block-cyclic partitioning with g = Grain

		IBM SP2		Origin 2000		Cray T3E	
Grain	Proc.	Time	Speedup	Time	Speedup	Time	Speedup
2	2	75.38	1.17	44.68	0.64	16.45	1.19
2	4	37.77	2.34	22.49	1.27	8.65	2.27
2	8	19.01	4.64	11.39	2.50	4.48	4.38
2	16	9.93	8.89	6.04	4.72	2.36	8.31
5	2	73.16	1.21	43.77	0.65	15.46	1.27
5	4	36.70	2.41	22.00	1.29	7.98	2.46
5	8	18.53	4.76	11.10	2.56	4.12	4.76
5	16	9.36	9.43	5.62	5.06	2.21	8.88
10	2	72.62	1.22	43.49	0.65	15.65	1.25
10	4	36.46	2.42	21.80	1.31	7.89	2.48
10	8	18.32	4.82	10.97	2.60	4.20	4.66
10	16	12.24	7.21	6.60	4.31	2.39	8.19
20	2	72.21	1.22	43.35	0.66	15.23	1.29
20	4	36.33	2.43	21.71	1.31	7.95	2.46
20	8	21.78	4.05	13.05	2.18	4.65	4.21
20	16	14.73	5.99	8.70	3.27	3.05	6.43

suitable kernel to build general parallel tools for Dynamic Programming like the described in the former section. Based in the proposed algorithms, a general parallel tool for Dynamic Programming has been built. This tool comes to fill a remarkable absence of such a kind of tool both in the sequential and parallel fields.

Acknowledgments

We wish to thank to the Centre Europeu de Paral-lelisme de Barcelona, to the Centre de Supercomputacio de Catalunya and to the CIEMAT for allowing us the access to their machines.

Table 4. Times and Speedup for llpdp code using block-cyclic partitioning with g = Grain

		IBM SP2		Origin 2000		Cray T3E	
Grain	Proc.	Time	Speedup	Time	Speedup	Time	Speedup
2	2	70.15	1.26	44.63	0.64	23.91	0.82
2	4	35.11	2.51	22.26	1.28	12.11	1.62
2	8	17.78	4.96	11.29	2.52	6.26	3.13
2	16	8.64	10.22	5.47	5.20	3.11	6.31
5	2	68.03	1.30	43.00	0.66	23.81	0.82
5	4	34.27	2.58	21.56	1.32	12.19	1.61
5	8	17.26	5.12	10.87	2.62	6.20	3.16
5	16	8.72	10.13	5.48	5.20	3.26	6.01
10	2	67.93	1.30	42.85	0.66	23.92	0.82
10	4	33.96	2.60	21.47	1.33	12.06	1.63
10	8	17.10	5.16	10.80	2.64	6.31	3.11
10	16	11.05	7.99	6.20	4.59	3.33	5.88
20	2	67.42	1.31	42.60	0.67	23.86	0.82
20	4	33.89	2.61	21.39	1.33	12.22	1.60
20	8	20.30	4.35	12.80	2.22	6.98	2.81
20	16	13.26	6.66	8.17	3.48	4.26	4.60

References

1. Aleksandrov V.. *Dynamic Programming Approach of the Knapsack Problems for the Ring of Processors*. WPDP93. Sofia. Bulgaria. 1993.
2. Almeida F., Morales D., Roda J.L., Rodríguez C., García F.. *The Single Resource Allocation Problem: Parallel Algorithms on Distributed Systems*. Proceedings of the 19th Technical Conference of the World Occam and Transputer User Group (WOTUG). Nottingham. 1996.
3. Almeida F., García F., Morales D., Rodríguez C.. *A parallel algorithm for the integer knapsack problem for pipeline networks*. Journal of Parallel Algorithms and Applications 6 (3/4). 1994.
4. Andonov, Raimbault, Quinton. *Dynamic Programming Parallel Implementations for the Knapsack Problem*. Technical Report 740. IRISA. June 1993.
5. Bellman R.. *Dynamic Programming*. Princeton U. P.. 1957.
6. Chen G., Jang J.. *An Improved Parallel Algorithm for 0/1 Knapsack Problems*. Parallel Computing. 18. 811-821. 1992.
7. Burns A., Friedman F.. *Parallel Ordered Enumerative Algorithms for the Multidimensional Knapsack Problem on Transputer Networks*. Proceedings of the Seventh Conference of the North American Transputer Users Group (NATUG 7). Atlanta. 116-132. IOS Press. 1995.
8. Edmonds P., Chu E., George A.. *Dynamic Programming on a Shared Memory Multiprocessor*. Parallel Computing. 19, 9-22. 1993.

9. Galil Z., Park K..*Parallel Algorithms for Dynamic Programming Recurrences with more than O(1) Dependency*. Journal of the Parallel and Distributed Computing. 21, 213-222. 1984.
10. Gibbons A., Rytter W., *Efficient Parallel Algorithms*. Cambridge University Press. 1988.
11. Hopcroft J., Ullman J.. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley Publishing Company. 1939.
12. Ibaraki T.. *Enumerative Approaches to Combinatorial Optimization, Part II*. Annals of Operations Research. Volume 11, N. 1-4, 1988.
13. Ibaraki T., Katoh N.. *Resource Allocation Problems. Algorithmic Approaches*. The MIT Press. Cambridge, Massachusetts. London. 1988.
14. JáJá J. *An Introduction to Parallel Algorithms*. Addison Wesley. 1992.
15. Karp R., Held M.. *Finite-State Processes and Dynamic Programming*. SIAM J. Appl. Math. 15, 693-768. 1967.
16. Kindervater G., Trienekens H.. *Experiments with Parallel Algorithms for Combinatorial Problems*. European Journal of Operational Research. 33, 65-81. 1988.
17. Lee J., Shragowitz E., Sahni S.. *A hypercube algorithm for the 0/1 knapsack problem*. Journal of Parallel and Distributed Computing 5, pp. 438-456.
18. Louka B., Tchuente M.. *Dynamic Programming on 2D Systolic Arrays*. Parallel Processing. 265-275. 1988.
19. Moldovan D., Fortes J.. *Partitioning and Mapping Algorithms into fixed size Systolic arrays*. IEEE Trans. Comput. C-35 (1), 1-12. 1986.
20. Morales D., Roda J., Almeida F., Rodríguez C., García F.. *Integral Knapsack Problems: Parallel Algorithms and their Implementations on Distributed Systems*. Proceedings of the 1995 International Conference on Supercomputing. 218-226. ACM Press. 1995.
21. Morin T., Marsten R.. *Branch-and-Bound Strategies for Dynamic Programming*. Operations Research. Vol. 24. 4, 611-627. 1976.
22. Myoupo J.. *Mapping Dynamic Programming onto Modular Linear Systolic Arrays*. Distributed Computing. 6, 165-179. 1993.
23. Teng S.. *Adaptative parallel algorithms for integral Knapsack Problems. Journal of Parallel and Distributed Computing*. 8, 400-406. 1990.
24. Syslo M., Deo N., Kowalik J.. *Discrete Optimization Algorithms with Pascal Programs*. Prentice-Hall. Eglewood Cliffs. New Jersey. 1983.
25. Ulm D., Wang P.. *Solving a two-dimensional Knapsack Problem on SIMD Computers*. International Conference on Parallel Processing. 1992.
26. Wah B., Li G., Fen C.. *Multiprocessing of Combinatorial Search Problems*. Computer. Vol. 18. 6, 93-108. 1985.